SCIMITAR: Scalable Stream-Processing for Sensor Information Brokering

Kurt Rohloff, Jeffrey Cleveland, Joseph Loyall
Raytheon BBN Technologies
Cambridge, MA, USA
{krohloff,jcleveland,jloyall}@bbn.com

Timothy Blocher
US Air Force Research Laboratory
Rome, NY, USA
Timothy.Blocher@us.af.mil

***Abstract*—Current sensor collection capabilities produce an incredible amount of data that needs to be processed, analyzed, and distributed in a timely and efficient manner. Information Management (IM) services supporting a publish-subscribe and query paradigm can be a powerful general purpose approach to enabling this information exchange between decoupled and dynamic information producers and consumers. These IM services will only be of value, however, if they can support operations in a manner that is responsive to the sheer quantity and frequency of data produced by surveillance platforms. Cloud computing is the technology of choice for providing the resources and services needed to enable and mange large-scale distributed computation. To date, there has been little work to develop highly scalable, dynamic IM processing and dissemination services in a cloud computing environment. In this paper we discuss our design, implementation and evaluation of a prototype cloud-based information broker which is a critical component of a highly scalable, distributed IM System. The brokering prototype is designed using a distributed stream processing framework and is shown to scale nearly linearly with the number of computing nodes as information load and subscription quantity increases.**

Keywords—sensor, cloud computing, stream computing, publish-subscribe, information brokering.

I. INTRODUCTION

Real-time and near-real-time information sharing continues to increase in importance for many military, security and intelligence domains. This has led to challenges in information management that has been exacerbated by the proliferation of increasingly capable sensors. The volume of available sensed information and the consumers of this information have led to the following two challenges:

- How to distribute large amounts and high rates of sensor information to the proper consumers without burying the consumers in extraneous information.
- How to process and filter content-rich information destined for consumers at scale and high speeds.

Publish-subscribe *brokering* has emerged to solve the former of these challenges. Brokering is the task of matching incoming published data against a set of subscriptions, which is the core functionality of publish-subscribe systems. Within such systems, consumers register subscriptions and publishers, such as sensors, publish data. The information broker matches published information with subscriptions, and forwards the published information to subscribed consumers.

In recent years a canonical problem in information brokering for has been to develop approaches that can scale to handle large volumes of information generated by sensor networks. Unfortunately, up until now, there have been few capabilities to sufficiently scale key information brokering capabilities to support the increasingly large volumes of information generated by sensor networks. Information brokering can be computationally expensive and its cost can increase non-linearly with the number of subscribers and the complexity of their subscriptions. Additionally, information brokering needs to be able to handle large amounts of aperiodic and periodic input data of heterogeneous sizes and data formats in real time or near-real time.

Cloud computing has emerged as a powerful medium for affordable large-scale computing which could be used to address the brokering problem. Cloud computing provides infrastructure access, software licensing, training and maintenance bundled into large commodity computing data centers that support elastic resource allocation. Cloud computing could help enable broader distributed information interactions built upon the publish-subscribe model with real-time and network-centric operational requirements. To date, however, little has been done to support publish-subscribe in cloud environments, despite the advantage of enabling scalable brokering at high speed larger numbers of clients, and more complex filtering and brokering. In this paper we describe our design, development, experimentation and analysis of SCIMITAR, a cloud-based information brokering capability implemented using the Storm Stream Processing Framework [16].

The remainder of this document is organized as follows. In Section II we discuss the needs for high-performance brokering. In Section III we discuss stream processing technologies which we use as the basis for our SCIMITAR prototype. In Section IV we present our overall design and implementation details for our information brokering concept. In Section V we discuss our experimental evaluation of our cloud-based brokering prototype. In Section VI discuss conclusions and possible future work.

II. THE INFORMATION BROKERING CHALLENGE

Eugster et al [7] provides an overview of the pub-sub interaction pattern, highlighting the decoupled nature of publishers and subscribers in time, space, and synchronization. There have been multiple prior approaches to publish-subscribe systems including the OMG Data-Distribution Service (DDS) Specification [19] and JMS [20]. For the common pub-sub brokering capabilities we are attempting to improve upon,

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **NOV 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE **SCIMITAR: Scalable Stream-Processing for Sensor Information Brokering**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Raytheon BBN Technologies,10 Moulton Street,Cambridge,MA,02139**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Military Communications Conference (MILCOM), November 18-20, 2013, San Diego, CA, pp. 1856-1861.

14. ABSTRACT
Current sensor collection capabilities produce an incredible amount of data that needs to be processed, analyzed, and distributed in a timely and efficient manner. Information Management (IM) services supporting a publish-subscribe and query paradigm can be a powerful general purpose approach to enabling this information exchange between decoupled and dynamic information producers and consumers. These IM services will only be of value, however, if they can support operations in a manner that is responsive to the sheer quantity and frequency of data produced by surveillance platforms. Cloud computing is the technology of choice for providing the resources and services needed to enable and mange large-scale distributed computation. To date, there has been little work to develop highly scalable dynamic IM processing and dissemination services in a cloud computing environment. In this paper we discuss our design implementation and evaluation of a prototype cloud-based information broker which is a critical component of a highly scalable distributed IM System. The brokering prototype is designed using a distributed stream processing framework and is shown to scale nearly linearly with the number of computing nodes as information load and subscription quantity increases.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**6**	

consumers register subscriptions based on attributes (i.e., *metadata*) about the information and its contents. Published information objects are matched by the broker against the registered subscription predicates. The broker tags each IO with the endpoints to which it is to be delivered. The brokered object is then passed to a dissemination service which transmits the object to its subscribers. Information is then disseminated to each consumer only those information objects that match the consumer's interests. We utilize explicit submission and dissemination services for receiving published information and transmitting brokered information to subscribers.

We focus on supporting published information represented as an Information Object (IO) consisting of XML metadata and a payload. This representation is fairly standard for information management systems. The metadata contains details such as publisher ID, timestamps, source location, payload format, and attributes over which interest can be registered. The form of the payload will be dictated by the publisher, for instance cameras would likely push out binary imagery data in a payload.

The subscriptions can be complex, and the brokering services we focus on support brokering that can match over rich sets of metadata using the XPath language to represent subsubscription expressions [21] which are to be matched against the metadata of published IOs. XPath is a rich expression language and can be computationally expensive. Computational cost can grow non-linearly with the number of predicates in an XPath expression, and metadata on published data may be very large.

A key difference between our information brokering capability and many others is support for richer "matching" of brokered information with subscriptions. DDS, for example, utilizes a *topic* based subscription model in which published data contains a topic field and subscribers who have registered for that specific topic will receive the data. This topic-based subscription model is a coarse division of all information into logical groups based on a shared topic name.

Challenges in supporting a scalable brokering service in a cloud computing environment at large scales include that the service must be able to (1) scale to the throughput of published data objects (2) scale with the quantity and complexity of registered subscription predicates and published metadata, and (3) maintain up-to-date and consistent views of registered subscriptions. These challenges are related to the number of publisher and subscriber clients. An individual publisher may create much more data than other publishers and similarly any individual subscriber may register more or more complex subscriptions than another subscriber.

There has been surprisingly little related work in the application of cloud computing technologies to the problem of information brokering. Apache Kafka [5] is a distributed and scalable commit log service which provides brokering capabilities using a topic-based subscription model. Prior research in scalable brokering has focused on challenges of geographically distributed multicast [6][7]. G-QoSM [2] describes a mechanism for QoS management in a grid computing architecture by reserving some of the system capacity for utilization by certain classes of operations if there is resource failure or congestion. This elasticity is similar to the benefits we provide in our SCIMITAR approach, but through a different underlying brokering mechanism that accounts more explicitly for low-level resource allocation. A key benefit of our SCIMITAR approach is that it is easier to use than these prior approaches, to better fit the elastic resource allocation model encouraged by cloud providers. Kang et al provides an approach for resource provisioning in stream-based services but without publish-subscribe capabilities [13]. Tudoran et al describes an approach to streaming data analysis, but without publish-subscribe capabilities [17]. Zhu et al [22] provides an approach to cloud-based publish-subscribe that does not provide the breadth of capabilities provided in SCIMITAR.

III. Cloud Computing and Stream Processing

When designing cloud-based brokering capability, we considered the design of a distributed computing architecture as the largest challenge. This architecture needs to support the desirable features of cloud computing including horizontal scalability and resilience to individual computing node failures. Horizontal scalability is the ability of a system to provide increased performance as more computational resources are available. The distributed computing architecture needs to enable low latency and "exactly-once" delivery of published information to subscribers as required for practical Pub-Sub middleware. By "exactly once" delivery we mean that the brokering capability should not deliver replicated information to a subscriber.

In order to support these requirements we focus our design and development on stream processing based frameworks.

Based on a cursory review of the current cloud computing distributed computing paradigms, one might consider use any of the highly scalable batched Map-Reduce technologies as, for example, implemented in Hadoop [10]. Although extremely scalable for information processing, this approach cannot provide a scalable, low-latency approach to information. Hadoop needs to register information in the Hadoop NameNode service, and then read from disk for any brokering function that could be supported by Hadoop. Whereas successful uses of batched MapReduce support iterative estimation of parameters for information pulled from disk, our needs for brokering on IOs already stored in memory do not align well with Hadoop.

We utilize an alternative and increasingly popular paradigm called stream processing to develop a cloud-based information brokering prototype. In the stream processing framework, "flows" of data are ingested and the stream processing framework uses parallel computation to process this data across multiple compute nodes.

Until recently, most stream computing frameworks have been proprietary, such as IBM's Infosphere Streams [18]. Recently emerging open-source stream processing frameworks include Apache S4 [15] and Storm [16]. These two open-source streaming frameworks provide near-real-time response to input data while also enabling horizontal scalability. Storm provides fault-tolerance guarantees on data processing despite occurrences of failures of individual computing nodes. Storm is designed to restart failed computation in response to partial failures by maintaining limited state information. We selected Storm as the basis for our SCIMITAR brokering capability due to Storm's general computing model and support for guaranteed processing of published information.

In the Storm framework, streams of data are unbounded lists of tuples. *Spouts* represent the data sources that emit possibly multiple streams. *Bolts* perform the computation to take streams and convert them to output streams. A Storm topology is a network of bolts connected by flows of streams and streams emanating from possibly multiple spouts.

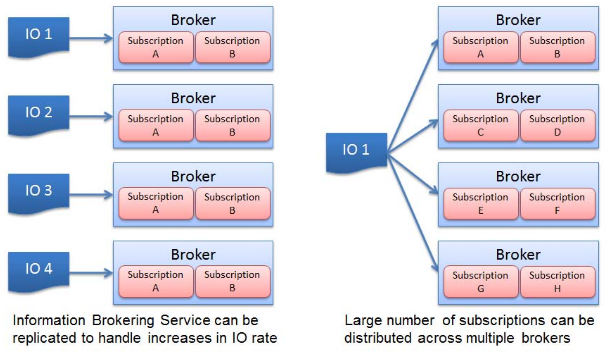
IV. DESIGN AND IMPLEMENTATION

To scale with data throughput, subscriptions must be replicated across multiple broker processes in a manner that allows more brokers to be spawned for higher throughput situations and each published IO to be dispatched to one of these brokers. To scale with the number of subscriptions, subscriptions must be distributed across multiple broker processes, and published IOs must be passed to each of these brokers to ensure that it is compared against every active subscription. These two situations are illustrated in Figure 1.

To provide this functionality, SCIMITAR distributes subscriptions across multiple *subscription groups*. Each subscription group is responsible for matching published information against an exclusive subset of all subscriptions. Within each subscription group is a set of replicated brokering processes that compare published information against the subscriptions assigned to its group. By passing each published IO to a single brokering process within every subscription group, each IO is matched against every active subscription.

All output from the subscription groups is sent to a set of parallel filters which remove duplicates and forward brokered IOs to the dissemination service. The specific filtering process used is based on a hash of each IO's unique ID ensuring the same filter sees all of the endpoints for a given IO. Additionally, this processing layer caches which consumer endpoint an IO is being delivered to and filter out any duplicates.

This design allows us to meet our two scalability goals 1) increasing the number of brokering processes within each subscription group to scale with increased information throughput and 2) increasing the number of subscription groups to scale with a larger quantity of subscriptions or more computationally complex subscriptions.



IO 1
IO 2
IO 3
IO 4
Broker
Subscription A
Subscription B
Information Brokering Service can be replicated to handle increases in IO rate
IO 1
Subscription C
Subscription D
Subscription E
Subscription F
Subscription G
Subscription H
Large number of subscriptions can be distributed across multiple brokers

Figure 1 Parallelization of Brokering

To support our third design goal of maintaining up-to-date and consistent views of registered subscriptions we implemented a persistent globally accessible database store. Subscription modification processes listen for commands from clients to add/remove/update subscriptions and then update the subscription store and send update commands to the appropriate subscription group. As brokering processes within a subscription group start, they query the Subscription Store with its Group ID to receive all current subscriptions.

In the context of the Storm framework, we designed Broker Bolts which operate as the brokers. Internally, the brokering bolt iterates over subscriptions for each IO received and immediately outputs matches. The output of the broker bolts is fed to the Filter/Output Bolts. The filters contain a TimeCacheMap of IO UID to List of Endpoints. Field Grouping on IO UID ensures that the same worker gets each endpoint for a given UID to reduce the likelihood of a subscriber receiving duplicate IOs, but there are still cases where duplicates might be delivered. A Field Grouping uses a specific Field of the tuple to identify which downstream task that tuple should be assigned to. In other words, the Field Grouping on IO UID means every tuple where IO UID is set to "A" will be delivered to the same task. If a worker dies then a different worker may be assigned that UID and duplicates may be delivered. Also in our design if the processing of an IO takes too long then its entries may have been removed from the cache and duplicates may be delivered.

We developed a Subscription Modification Bolt to manage subscription updating. The Subscription Modification Bolt communicates with the Subscription Store, which is implemented using a PostgreSQL DB. This database stores the details of subscriptions, as well as to which subscription group each is assigned. A schematic of this occurring is shown in Figure 3 where specialized subscription modification spouts in Storm receive subscription update tasks which are routed to the appropriate subscription groups and subsequently to brokers.

Input into the system is provided via a distributed buffer implemented using Kestrel [14]**.** The IO spout polls the buffer for published IOs, while the Subscription Modification spout polls a second set of buffers for commands to add/remove/update subscriptions. The output bolt places the brokered IO into a similar output buffer, which test clients continuously poll. A schematic of this architecture is shown in Figure 2. In a deployed system, a dissemination service would perform this task and then deliver the IO to the correct endpoints.

V. EXPERIMENTAL EVALUATION

To evaluate the performance of our cloud-based brokering prototype, we conducted experiments that evaluated correctness and scalability with respect to throughput as a function of the number of Storm Supervisor nodes and subscriptions. We evaluate scalability in terms of the rate of published IOs and the number of registered subscriptions. We performed two sets of scalability experiments:

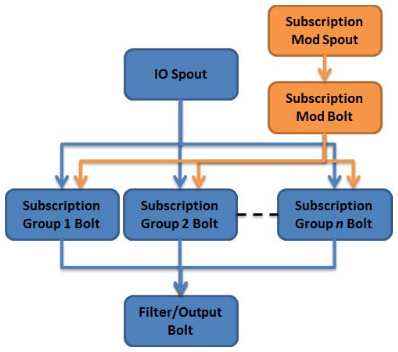

Subscription Mod Spout
IO Spout
Subscription Mod Bolt
Subscription Group 1 Bolt
Subscription Group 2 Bolt
Subscription Group *n* Bolt
Filter/Output Bolt

Figure 3 Storm Topology

1. IO Input Rate Scalability to measure how IO brokering throughput changes as a function of IO input rate and the number of compute nodes used for brokering.
2. IO Subscription Scalability to measure how IO brokering throughput changes as a function of the number of registered subscriptions and compute nodes used for brokering.

Additionally, we have results from experiments indicating that subscription groups are a feasible way to minimize latency.

A. Experimentation Environment

We developed a custom deployment infrastructure and a test framework to support experimentation on our cloud-based brokering services. The basis of our deployment framework is the capability to create, populate, configure, and run the VMs necessary for our cloud-based brokering capability. This framework maps the requirements of our prototype cloud services to "roles," each containing requirements for software and a VM and each containing instructions on how to configure the software and system to interact with the other nodes in the prototype services cloud. We wrote this automated deployment infrastructure with the *boto* API [1]. Boto provides an interface to several Infrastructure as a Service (IaaS) cloud frameworks including Amazon Web Services and Eucalyptus. For load testing, we used *The Grinder* [9], a Java load testing framework that facilitates running a distributed test using many load injector machines.

Our experimental framework consists of virtual machines providing the following roles: publisher and subscription clients running Grinder agents, Kestrel queues serving as input and output buffers, nodes running ZooKeeper and a node running the Storm Nimbus service to coordinate the Storm topology, a node serving as a Grinder console to coordinate the test clients, a node running PostGreSQL to provide persistent subscription storage, and nodes running Storm Supervisors which we refer to as *worker nodes* and which were responsible for the brokering computation.

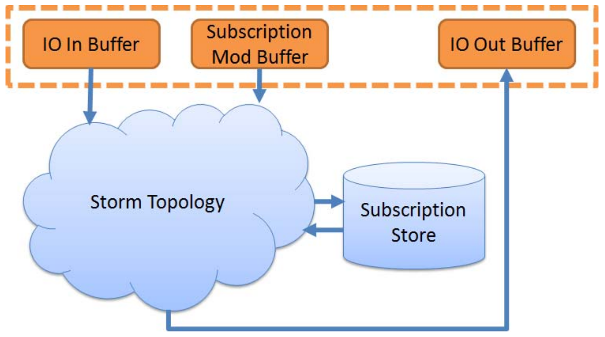


Figure 2 SCIMITAR Architecture

B. Experiment Execution to Investigate Scalability from adding more Worker Nodes for Brokering

After deployment, we ran an initial experiment to collect data on how IO throughput performance varies with the number of brokering nodes while supporting either 64 or 128 subscriptions. We ran this initial set of experiments to verify that the effect of scaling the number of nodes would be consistent across different numbers of registered subscriptions. Figure 4 shows the graph from one of the 128 subscriptions runs, the one with 32 nodes. For these experiments the published IOs contained 1 KB payloads and metadata which contained less than 20 lines of XML.

Table 1 presents the collected data which relates the maximum brokering throughput in IOs per second as a function of the number of brokering worker nodes and subscriptions. Given the variability in this measurement over time, we collected 2 significant digits of data. This data is also shown as a graph in Figure 4.

As can be seen from Figure 4, throughput appears to grow linearly with the number of worker nodes used for brokering. This is the behavior we expected and is highly desirable as it provides a simple approach to scale our brokering prototype to deal with high throughput by adding more brokering compute nodes. This linear scaling shows that we can deal with increasingly large throughput requirements by adding extra computing nodes as brokering workers.

We used the standard linear regression technique to estimate the slope of the lines which we also plot in Figure 4. The slope of the 64 subscription throughput best-fit line (56 IOs per sec. per node) is almost double the slope of the 128 subscription throughput best-fit line (32 IOs per sec. per node.) These measurements indicate that the brokering throughput may be close to inversely proportional to the number of subscriptions.

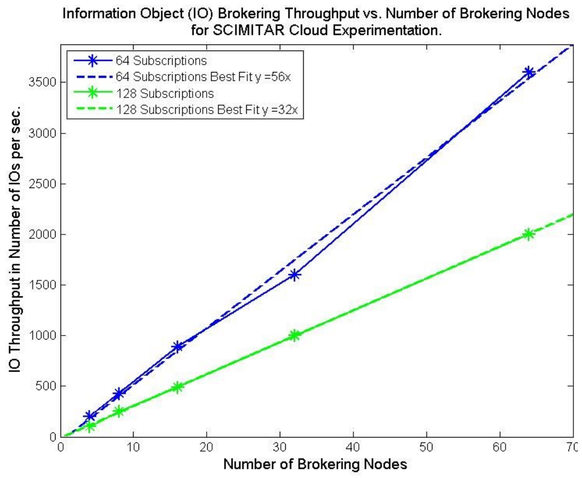


Figure 4. IO Throughput with Best-fit Lines

Table 1. IO Throughput as a function of the number of brokering compute nodes for two different numbers of registered subscriptions.

Number of Brokering Nodes	64 Subscriptions	128 Subscriptions
4	200	100
8	430	250
16	890	490
32	1600	1000
64	3600	2000

This near-inverse-proportional behavior is expected based on our design in that it intuitively takes longer to broker over more subscriptions. We investigate the inverse proportional relationship further in an experiment set below, and we provide a more detailed analysis of the relationship between throughput and the number of subscriptions.

We did not explore the far upper limits of the linear scalability of IO throughput with respect to the number of worker nodes used for brokering. Our hypothesis is that there are limits to how far our experimental cloud-based brokering framework can scale based either on the underlying infrastructure, inter-node communication bottlenecks, or the stream processing framework. An exploration of the upper limits of the scalability of our framework is an area of ongoing and future work.

Although we did not analyze how scalability varies based on IO size, our expectation is that IO size will not greatly affect scalability, but there is likely to be an upper limit on the capability of our experimental framework to handle very large multi-gigabyte IOs at very high throughputs.

Because the suspected over-flow effect occurs only when the input rate approaches the maximum brokering rate, and because the maximum brokering rate is highly linear with respect to the number of computing nodes, we believe it is feasible to predict when the buffer-overflow will occur. As such, the experimental observations in Figure 4 imply that it should be straightforward to select the number of brokering compute nodes that should be allocated in a cloud for our brokering prototype to successfully process desired brokering throughputs.

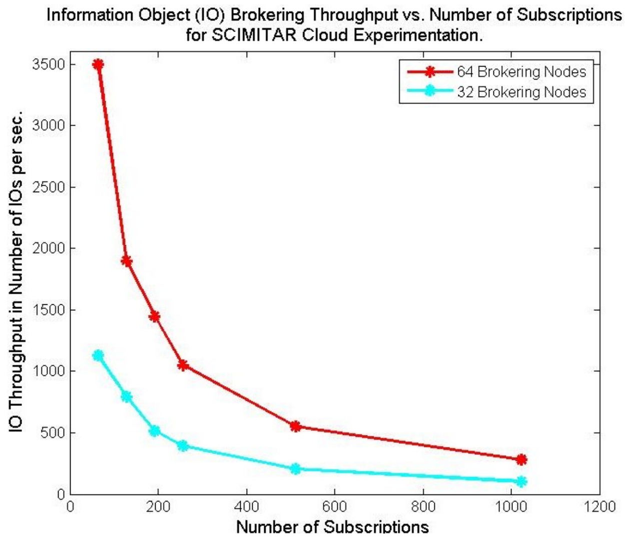


Figure 5. IO Throughput as a Function of Number of Subscriptions.

Table 2. IO throughput as a function of the number of registered subscriptions for two different numbers of brokering compute nodes.

Number of Subscriptions	Throughput (IOs/second) 32 Nodes	Throughput (IOs/second)64 Nodes
64	793	3500
128	515	1900
256	393	1450
512	204	1050
1024	104	550

C. Experiment Execution to Investigate Scalability from adding Subscriptions

As with the experiment set that analyzes the impact of adding more worker nodes for brokering, we collected data on the maximum brokering throughput as a function of the number of subscriptions for various brokering worker node configurations. Table 2 presents the maximum brokering throughput in numbers of IOs per second as a function of the number of brokering compute nodes and subscriptions. This data is also shown as a graph in Figure 5.

Our experimental data indicates that the brokering throughout is inversely proportional to the number of subscriptions. This behavior is expected as it intuitively takes longer to broker over more subscriptions. To investigate the relationship between IO throughput and the number of subscriptions in our brokering prototype, we plotted this relationship in a log-log plot shown in Figure 6.

We used the standard linear regression approach to find best-fit lines for the experimental data as shown in Figure 6. We can map lines in the log-log graph back to the linear-linear graph to identify the relationship between the throughput and the number of subscriptions. We start from the following equation which represents how our experimentally identified linear relationship relates throughput and the number of subscriptions in the log-log domain:

$$\log y = m \log x + b$$

where

- y represents throughput
- x represent the number of subscriptions
- m and b are parameters that determine relationship between throughput and subscriptions

We convert this equation to a relationship between how our experimentally identified linear relationship relates throughput and the number of subscriptions in the linear-linear domain:

$$y = \frac{10^b}{x^{-m}}$$

As can be seen in Figure 6, the slopes of the linear regression lines for the experimental data are very close for the 32 node and 64 node cases. For the 32 brokering compute node case the exponent -m is 0.89 and the 64 brokering compute node case the exponent -m is 0.91.

Our experimental results show that increases in the number of brokering compute nodes offsets increases in the number of subscriptions to maintain throughput. Throughput grows linearly with the number of brokering compute nodes, but sub-linearly with the reciprocal of the number of subscriptions. As such, if our prototype needs to maintain a level of throughput despite an increase in the number of subscriptions, we would need to add a relatively smaller number of extra brokering compute nodes to maintain throughput. This shows our prototype design can scale effectively with increases in subscriptions by adding more compute nodes.

All of the data collected, presented and discussed until now on brokering performance was derived from experiments conducted on an Amazon cluster. We partially recreated our experiments on an internal Eucalyptus cluster which we could not scale as large as the Amazon environment due to a lack of computation resources. We recreated our subscription scalability experiments on the relationship between throughput and the number of subscriptions for 8 and 16 brokering compute node setups in our private cloud. The results on our internal cluster indicate a best-fit linear regression in the log-log domain with –m=0.97.

VI. Conclusions and Future Work

Our research, prototype development, and evaluation have established a firm basis for cloud-based IM systems. It has shown that brokering of IOs for client requests can be architected and implemented to work with emerging cloud-based streaming platforms and that, in doing so, publish-subscribe operations can be performed in the cloud at high speeds and at scale, both in terms of number of published objects (throughput) and in terms of number of registered subscriptions. We have shown that there are several existing and emerging cloud-based technologies that show the promise of being applied to IM service concepts and that provide varying levels of benefit for IM operations in the cloud.

A next step is to design and develop cloud-based versions of additional IM services to round out and complement our brokering capability. Messaging services such as HornetQ [8] and Kestrel [14] are promising candidates for supporting submission and dissemination. A wide range of distributed databases, e.g., Accumulo [3], Cassandra [4], and MongoDB [12], could be leveraged to back archive and query services.

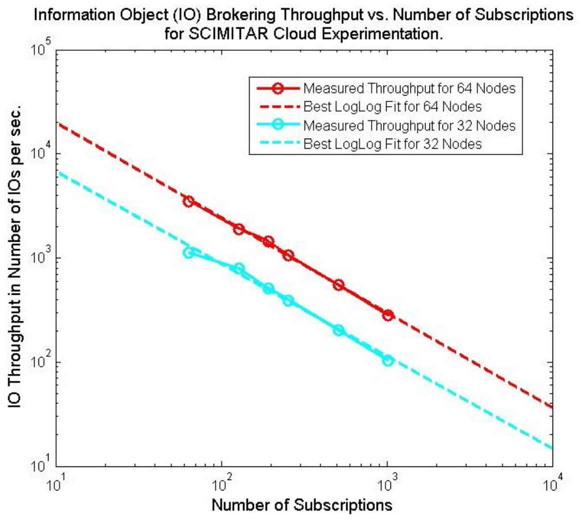


Figure 6. IO Throughput as a Function of Number of Subscriptions with Best-fit Linear Regression Lines.

Acknowledgements

This work was supported by the U.S. Air Force Research Laboratory under Contract Number FA8750-12-C-0083.

References

[1] Amazon Boto. Retrieved from http://boto.s3.amazonaws.com/index.html

[2] Al‐Ali, Rashid, et al. "An approach for quality of service adaptation in service‐oriented Grids." *Concurrency and Computation: Practice and Experience* 16.5 (2004): 401-412.

[3] Apache Accumulo. Retrieved from http://accumulo.apache.org/

[4] Apache Cassandra. Retrieved from http://cassandra.apache.org/

[5] Apache Kafka. Retrieved from http://kafka.apache.org/

[6] Banavar, G.; Chandra, T.; Mukherjee, B.; Nagarajarao, J.; Strom, R.E.; Sturman, D.C.; , "An efficient multicast protocol for content-based publish-subscribe systems," *Distributed Computing Systems, 1999. Proceedings*, 1999

[7] Eugster, Patrick Th, et al. "The many faces of publish/subscribe." *ACM Computing Surveys (CSUR)* 35.2 (2003): 114-131.

[8] Giacomelli, Piero. HornetQ Messaging Developer's Guide. Packt, 2012.

[9] Grinder. Retrieved from http://grinder.sourceforge.net/.

[10] Hadoop. Retrieved from http://hadoop.apache.org/

[11] HStreaming. Retrieved from http://www.hstreaming.com

[12] MongoDB. Retrieved from http://www.mongodb.org/

[13] Seungwoo Kang, Youngki Lee, Sunghwan Ihm, Souneil Park, Su-Myeon Kim, and Junehwa Song. 2010. Design and Implementation of a Middleware for Development and Provision of Stream-Based Services. *2010 IEEE 34th Annual Computer Software and Applications Conference* (COMPSAC '10). 92-100.

[14] Kestrel. Retrieved from https://github.com/robey/kestrel

[15] S4. Retrieved from http://s4.io/

[16] Storm. Retrieved from https://github.com/nathanmarz/storm/wiki

[17] Radu Tudoran, Gabriel Antoniu, and Luc Bouge. 2013. SAGE: Geo-Distributed Streaming Data Analysis in Clouds. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum* (IPDPSW '13). IEEE Computer Society, Washington, DC, USA, 2278-2281.

[18] InfoSphere Streams. Retrieved from http://www-01.ibm.com/software/data/infosphere/streams/

[19] DDS Spec. Retrieved from http://www.omg.org/technology/documents/dds_spec_catalog.htm

[20] Java Messaging Service. Retrieved from http://www.oracle.com/technetwork/java/index-jsp-142945.html

[21] XPath Syntax. Retrieved from http://www.w3schools.com/xpath/xpath_syntax.asp

[22] Yuqing Zhu, Jianmin Wang, and Chaokun Wang. 2011. Ripple: A publish/subscribe service for multidata item updates propagation in the cloud. *J. Netw. Comput. Appl.* 34, 4 (July 2011), 1054-10